

Estruturas de Dados - ESP412

Prof^a Ana Carolina Sokolonski

Bacharelado em Sistemas de Informação
Instituto Federal de Educação, Ciência e Tecnologia da Bahia
Campus de Feira de Santana

carolsoko@ifba.edu.br

November 28, 2024

Algoritmos de Ordenação

Algoritmos de Ordenação

1 *InsertionSort*

2 *ShellSort*

3 *BubbleSort*

4 Referências

InsertionSort

Ordenação por Inserção (*InsertionSort*)

Dado um vetor v com n números, $v = \{a_1, a_2, a_3, \dots, a_n\}$.

Ordenação por Inserção (*InsertionSort*)

Dado um vetor v com n números, $v = \{a_1, a_2, a_3, \dots, a_n\}$.

O algoritmo de Ordenação por Inserção faz uma inserção ordenada dos números de tal forma que o vetor resultante, após a inserção, é um vetor ordenado $v = \{b_1, b_2, b_3, \dots, b_n\}$, onde $b_1 \leq b_2 \leq b_3 \cdots b_{(n-1)} \leq b_n$.

Ordenação por Inserção (*InsertionSort*)

Dado um vetor v com n números, $v = \{a_1, a_2, a_3, \dots, a_n\}$.

O algoritmo de Ordenação por Inserção faz uma inserção ordenada dos números de tal forma que o vetor resultante, após a inserção, é um vetor ordenado $v = \{b_1, b_2, b_3, \dots, b_n\}$, onde $b_1 \leq b_2 \leq b_3 \cdots b_{(n-1)} \leq b_n$.

O algoritmo de *InsertionSort* é eficiente para a ordenação de pequenas quantidades de elementos. Seu custo é $O(N^2)$ no pior caso e $O(N)$ no melhor caso. [Cormen et al. 2009]

Ordenação por Inserção (*InsertionSort*)

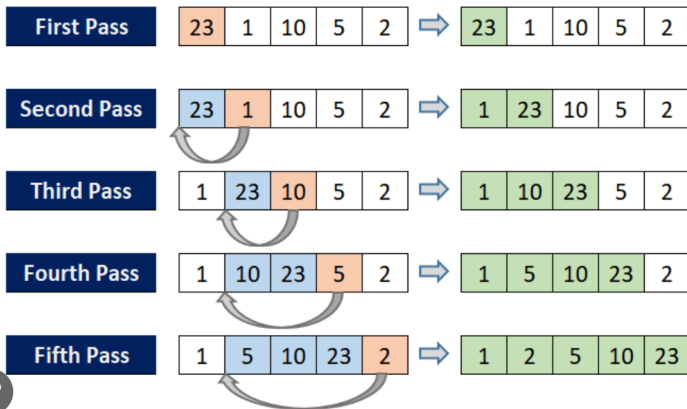
Dado um vetor v com n números, $v = \{a_1, a_2, a_3, \dots, a_n\}$.

O algoritmo de Ordenação por Inserção faz uma inserção ordenada dos números de tal forma que o vetor resultante, após a inserção, é um vetor ordenado $v = \{b_1, b_2, b_3, \dots, b_n\}$, onde $b_1 \leq b_2 \leq b_3 \cdots b_{(n-1)} \leq b_n$.

O algoritmo de *InsertionSort* é eficiente para a ordenação de pequenas quantidades de elementos. Seu custo é $O(N^2)$ no pior caso e $O(N)$ no melhor caso. [Cormen et al. 2009]

Vejamos um vídeo explicativo do algoritmo *InsertionSort* no Youtube: <https://www.youtube.com/watch?v=EdIKIf9mHk0>

Insertion Sort



Ordenação por Inserção → *InsertionSort*

Algorithm void insertionSort(*int tam*, *int v*[])

```
1: for  $i = 1$  to  $N$  do
2:    $j = i$ ;
3:   while ( $(j > 0)$  and  $(V[j] < V[j - 1])$ ) do
4:      $aux = V[j]$ ;
5:      $V[j] = V[j - 1]$ ;
6:      $V[j - 1] = aux$ ;
7:      $j --$ ;
8:   end while
9: end for
```

Ordenação por Inserção → *InsertionSort*

```
void insertionSort(int tam, int v[]){  
    int j,aux;  
    for(int i=1; i<tam; i++){  
        j = i;  
        while((j>0) && (v[j]<v[j-1])){  
            aux=v[j];  
            v[j]=v[j-1];  
            v[j-1]=aux;  
            j--;  
        }  
    }  
}
```

ShellSort

Ordenação por Inserção com Salto (*ShellSort*)

Proposto por *Donald Shell* em 1959. É uma extensão do *InsertionSort* com a intenção de sanar alguns problemas com o algoritmo de ordenação por inserção, que são:

Ordenação por Inserção com Salto (*ShellSort*)

Proposto por *Donald Shell* em 1959. É uma extensão do *InsertionSort* com a intenção de sanar alguns problemas com o algoritmo de ordenação por inserção, que são:

- Troca itens adjacentes para determinar o ponto de inserção.
- São efetuadas $(n - 1)$ comparações e movimentações quando o menor item está na posição mais à direita do vetor.
- Ineficiente para n grande.

Ordenação por Inserção com Salto (*ShellSort*)

Proposto por *Donald Shell* em 1959. É uma extensão do *InsertionSort* com a intenção de sanar alguns problemas com o algoritmo de ordenação por inserção, que são:

- Troca itens adjacentes para determinar o ponto de inserção.
- São efetuadas $(n - 1)$ comparações e movimentações quando o menor item está na posição mais à direita do vetor.
- Ineficiente para n grande.

O método de *Shell* contorna este problema permitindo trocas de registros distantes um do outro sem alto custo.

Ordenação por Inserção com Salto (*ShellSort*)

A proposta do algoritmo *ShellSort* é determinar um salto (h). Este salto é a distância entre os elementos que serão rearranjados.

Ordenação por Inserção com Salto (*ShellSort*)

A proposta do algoritmo *ShellSort* é determinar um salto (h). Este salto é a distância entre os elementos que serão rearranjados.

Os itens separados de h posições são ordenados: o elemento na posição x é comparado e trocado (caso satisfaça a condição de ordenação) com o elemento na posição $x + h$. Este processo se repete, decrescendo o h , até que $h = 1$, quando esta condição é satisfeita o algoritmo é equivalente ao método de inserção e o vetor estará totalmente ordenado.

Ordenação por Inserção com Salto (*ShellSort*)

A escolha do salto h pode ser qualquer sequência terminando com $h = 1$. Um exemplo é a sequência:

$$h(s) = 1, \text{ para } s = 1$$

$$h(s) = 3h(s - 1) + 1, \text{ para } s > 1$$

[D.E. Knuth 1973] provou experimentalmente que esta sequência é difícil de ser batida por mais de 20% em eficiência.

Ordenação por Inserção com Salto (*ShellSort*)

A escolha do salto h pode ser qualquer sequência terminando com $h = 1$. Um exemplo é a sequência:

$$h(s) = 1, \text{ para } s = 1$$

$$h(s) = 3h(s - 1) + 1, \text{ para } s > 1$$

[D.E. Knuth 1973] provou experimentalmente que esta sequência é difícil de ser batida por mais de 20% em eficiência. A complexidade deste algoritmo é desconhecida e o algoritmo não é estável.

Ordenação por Inserção com Salto (*ShellSort*)

A escolha do salto h pode ser qualquer sequência terminando com $h = 1$. Um exemplo é a sequência:

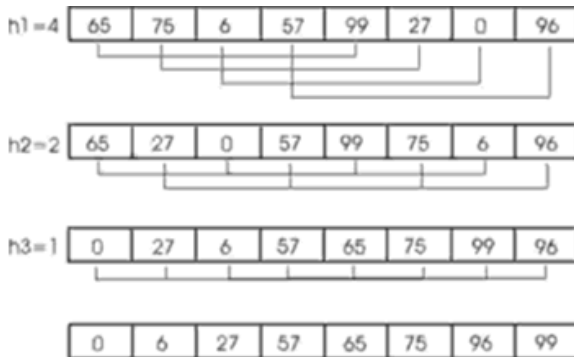
$$h(s) = 1, \text{ para } s = 1$$

$$h(s) = 3h(s - 1) + 1, \text{ para } s > 1$$

[D.E. Knuth 1973] provou experimentalmente que esta sequência é difícil de ser batida por mais de 20% em eficiência. A complexidade deste algoritmo é desconhecida e o algoritmo não é estável.

Vejamos um vídeo explicativo do algoritmo *ShellSort* no Youtube:
<https://www.youtube.com/watch?v=CmPA7zE8mx0>

Ordenação por Inserção com Salto → *ShellSort*



Algorithm void shellSort(*int* vet[], *int* tam)

```
1: salto = 1;
2: while (salto < tam) do
3:   salto = 3 * salto + 1;
4: end while
5: while (salto > 1) do
6:   salto / = 3;
7:   for i = salto to tam do
8:     valor = vet[i]; j = i - salto;
9:     while ((j ≥ 0) and (valor < vet[j])) do
10:      vet[j + salto] = vet[j]; j = j - salto;
11:    end while
12:    vet[j + salto] = valor;
13:  end for
14: end while
```

Ordenação por Inserção com Salto → *ShellSort*

```
void shellSort(int vet[], int tam) {  
    int i , j , valor;  
    int salto = 1;  
    while(salto < tam) {  
        salto = 3*salto+1;  
    }  
    while (salto > 1) {  
        salto /= 3;  
        for(i = salto; i < tam; i++) {  
            valor = vet[i];  
            j = i - salto;  
            while ((j >= 0) && (valor < vet[j])) {  
                vet [j + salto] = vet[j];  
                j -= salto;  
            }  
            vet [j + salto] = valor;  
        }  
    }  
}
```

BubbleSort

Ordenação por Comparação da Bolha (*BubbleSort*)

A ordenação por Comparação da Bolha (*BubbleSort*) é um algoritmo que realiza uma comparação local, par a par, dos elementos do vetor, e caso estes estejam desordenados, troca-os, e segue para o próximo par [R. Sedgewick and K. Wayne 2011].

Ordenação por Comparação da Bolha (*BubbleSort*)

A ordenação por Comparação da Bolha (*BubbleSort*) é um algoritmo que realiza uma comparação local, par a par, dos elementos do vetor, e caso estes estejam desordenados, troca-os, e segue para o próximo par [R. Sedgewick and K. Wayne 2011].

Repete este processo quantas vezes for necessário. Até que o algoritmo percorra todo o vetor sem realizar trocas. Neste momento, o vetor estará totalmente ordenado.

Ordenação por Comparação da Bolha (*BubbleSort*)

A ordenação por Comparação da Bolha (*BubbleSort*) é um algoritmo que realiza uma comparação local, par a par, dos elementos do vetor, e caso estes estejam desordenados, troca-os, e segue para o próximo par [R. Sedgewick and K. Wayne 2011].

Repete este processo quantas vezes for necessário. Até que o algoritmo percorra todo o vetor sem realizar trocas. Neste momento, o vetor estará totalmente ordenado.

Assim como o *InsertionSort*, o *BubbleSort* tem custo $O(N^2)$ no pior caso e $O(N)$ no melhor caso.

Ordenação por Comparação da Bolha (*BubbleSort*)

A ordenação por Comparação da Bolha (*BubbleSort*) é um algoritmo que realiza uma comparação local, par a par, dos elementos do vetor, e caso estes estejam desordenados, troca-os, e segue para o próximo par [R. Sedgewick and K. Wayne 2011].

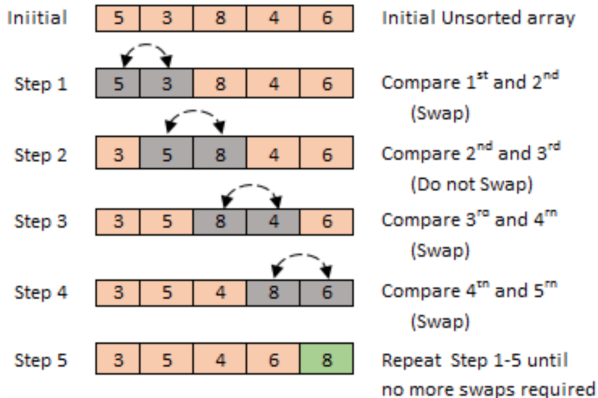
Repete este processo quantas vezes for necessário. Até que o algoritmo percorra todo o vetor sem realizar trocas. Neste momento, o vetor estará totalmente ordenado.

Assim como o *InsertionSort*, o *BubbleSort* tem custo $O(N^2)$ no pior caso e $O(N)$ no melhor caso. Vejamos um vídeo explicativo do algoritmo *BubbleSort* no Youtube:

<https://www.youtube.com/watch?v=Iv3vgjM8Pv4>

BubbleSort

Bubble sort example



Ordenação por Comparação da Bolha → *BubbleSort*

Algorithm void bubbleSort(*int tam*, *int v*[])




```
1: flag = true;
2: while flag do
3:   flag = false;
4:   for i = 1 to N do
5:     if  $V[i - 1] > V[i]$  then
6:       flag = true;
7:       aux =  $V[i]$ ;
8:        $V[i] = V[i - 1]$ ;
9:        $V[i - 1] = aux$ ;
10:    end if
11:  end for
12: end while
```

Ordenação por Comparação da Bolha → *BubbleSort*

```
void bubbleSort(int tam, int v[]){  
    int aux, flag = 1;  
    while (flag){  
        flag = 0;  
        for(int i=1; i<tam; i++){  
            if (v[i-1] > v[i]){  
                flag = 1;  
                aux=v[i];  
                v[i]=v[i-1];  
                v[i-1]=aux;  
            }  
        }  
    }  
}
```

Referências

Referências

-  CORMEN, T. H. et al. *Introduction to Algorithms*. 2nd. ed. [S.I.]: The MIT Press, 2009. ISBN 0262032937.
-  D.E. Knuth. *The Art of Computer Programming*. [S.I.]: Addison-Wesley, 1973. v. 1 - 3.
-  R. Sedgewick and K. Wayne. *Algorithms*. 4th edition. ed. [S.I.]: Addison-Wesley, 2011. v. 4.